AGENT-BASED COMPUTATIONAL GEOMETRY

Satine Paronyan

Term Report
CSS 600: Independent Study

Master of Science in Computer Science & Software Engineering

University of Washington Bothell
Spring 2020
Contents

1 Introduction ........................................................................................................................................... 3

2 Range Search ....................................................................................................................................... 3

2.1 Range Search Algorithms .................................................................................................................. 4

2.1.1 MASS Algorithm .......................................................................................................................... 4

2.1.2 MapReduce and Spark Algorithm ................................................................................................. 5

2.2. Results ............................................................................................................................................. 6

2.2.1 Programmability ............................................................................................................................ 6

2.2.2 Execution Performance ................................................................................................................... 7

3 Future Improvements ........................................................................................................................... 9

4 References .......................................................................................................................................... 9
1 Introduction

Computational geometry is a field of computer science that involves the design, analysis, and modeling of efficient algorithms to solve complex geometric problems. Computational geometry applied in computer graphics, geometric modeling, computer vision, geolocation, motion planning, and parallel computing. The computational geometry applications are computationally complex and involve large datasets. As a result, computational geometry algorithms require constant improvements and optimizations to achieve the best possible efficiency. Parallelization is an optimal solution to achieve an efficiency of computational geometry algorithms. MapReduce and Spark, which are the common big-data parallelization tools for computational geometry applications, are beneficial when used for data-streaming analysis tasks. The better approach is to use an agent-based parallelization approach, which fits better into an analysis of structured and geometric data. The focus of my research is the agent-based parallelization of the algorithms used to solve computational geometry problems.

The MASS library is a parallel-computing library developed by Professor Fukuda and DSL research group at the University of Washington Bothell. This library is used for multi-agent spatial simulations. The collective and emergent group behavior of agents such as propagation, swarming, colliding, and occasional repelling makes it easier to discover attributes of structured and geometric datasets.

The goal of my research is to demonstrate the efficiency of the agent-based parallelized algorithms with the MASS library, focusing on solving computational geometry problems. The plan is to solve complex computational geometry problems using many agents. Currently, several static problems, such as the closest pair of points, convex hull, Voronoi diagram, Delaunay triangulation, and shortest path are implemented by DSL. My research takes further the work on agent-based computational geometry by parallelizing four geometric and graph problems. This spring quarter, I developed the agent-based algorithm for the Range Search problem. In addition, I reviewed the traditionally used approaches for the range search problem, such as parallelization with MapReduce and Spark. Finally, compared the performance results of the three different parallelization approaches. Potential problems we look for future work include but are not limited to largest empty circle, point location, and minimum spanning tree.

2 Range Search

The range searching problem is one of the fundamental problems in computational geometry. Range searching has applications for range queries in databases, geolocation, data analysis, and statistics. Range searching is also often a subroutine step in the algorithms for solving more complex geometric problems. The range searching problem consists of preprocessing a set of \( N \) points in the plane to determine which points reside within a query rectangle (range). The query
range includes four values: x- minimum, maximum and y- minimum, maximum coordinates in a plane.

2.1 Range Search Algorithms

The baseline of the range searching algorithms is the construction of a multidimensional binary tree (KD tree). KD tree for two-dimensional points is a modified two-dimensional binary search tree (BST), which alternates x- and y- coordinates as a key for inserting elements. The alternating sequence starts with the x-coordinate. The construction of KD tree consists of recursively partitioning the plane into two halfplanes, where the point positioned at the bisection line is the next point to be inserted into the tree with respect to x and y dimensions. Each bisection line is determined after sorting the points by x or y coordinate depending on the next dimension of the KD tree level. The bisection line is determined by dividing the number of points by two. Because of using sorting, bisection lines, as well as x and y dimension alternations, the KD tree is built as a balanced BST.

2.1.1 MASS Algorithm

Algorithm 1: Range Search (MASS)

- Read input points (x, y) from input file
- Construct Point2D list from input points
- Sort the list by x value
- Number of Point2D’s is the number of vertices in GraphPlaces
- Create GraphPlaces with GraphPlaces(int handle, String className, int size), where size is the number of vertices that are split among computing nodes.
- Build KD Tree using GraphPlaces.
- Create an Agent at the root vertex.
- For each subtree where agents are created:
  while(root != null) do
    - Parent Agent check if its current vertex’s Point2D (x, y) is in the query range. If yes, save point into agent’s list of results.
    - Parent Agent check if its left and right children are not null. If both, left and right child are present, Parent Agent migrates to left node and Spawns a child Agent for
the right node. If only one child node exists Parent Agent migrates there without an additional child spawn.

```java
if (root.left == null && root.right == null) {
    - Parent Agent returns a result list of all the discovered points to the main program.
}
```

end if

end while

- At the main program:
  ```
  while Agents > 0 do
    - Collect all points sent by agents
  end while
  ```

2.1.2 MapReduce and Spark Algorithm

**Algorithm 1: Range Search (MapReduce and Spark)**

- Read input points \((x, y)\) from input file
- Partition points into small slices
- **for each** partition **do**
  - Sort points by \(x\) coordinate
  - Remove duplicate points if any
  - Build KD tree of points
  - Perform range search on KD tree. The search consists of traversing the tree with respect to \(x\) and \(y\) dimensions and determining the points within the query range.
  - Store the discovered points in a local list of search results.
  - Return discovered points from the result list.
- **end for each**
- Collect all search result points from each partition.
2.2 Results

2.2.1 Programmmability

The programmability of the Range Search implementations is measured by the following metrics:

- **Boilerplate code** – number of lines of code required to set up the environment
- **Lines of code** – total number of lines of code in the implementation
- **Number of classes** – total number of classes in the implementation

<table>
<thead>
<tr>
<th>Parallel Framework</th>
<th>Boilerplate code</th>
<th>Lines of code</th>
<th>Number of classes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MapReduce</strong></td>
<td>25</td>
<td>~ 450</td>
<td>6</td>
</tr>
<tr>
<td><strong>Spark</strong></td>
<td>6</td>
<td>~ 350</td>
<td>3</td>
</tr>
<tr>
<td><strong>MASS</strong></td>
<td>3</td>
<td>~ 490</td>
<td>6</td>
</tr>
</tbody>
</table>

The comparison of three different implementations of Range Search shows that MapReduce implementation consists of the highest number of boilerplate lines of code in contrast to Spark and MASS implementations (see Table 1). The implementation of Range Search using MapReduce consists of two MapReduce jobs that makes the most of the boilerplate code, where each job consists of setting up its dedicated mapper and reducer classes. The boilerplate code in Spark implementation consists of setting up Spark context and broadcasting query range values to the context. MASS requires the fewest boilerplate code to set up the environment. This boilerplate code consists of initializing MASS, setting up a debugging level, and shutting down MASS when the computation is finished.

The total number of lines in MapReduce and MASS implementations are relatively compatible. Yet, due to the needed custom agent and vertex place classes, the total number of lines is slightly higher than MapReduce implementation. Spark implementation requires the fewest number of lines of code, since it does not require a high quantity of boilerplate code. Another reason is that the Spark programming model benefits from having RDDs instead of separate Mapper and Reducer classes as in MapReduce or having custom Agent or Place classes as in MASS.

Finally, the fewest number of classes is required by Spark implementation, whereas MapReduce and MASS implementations require more classes. As noted previously, Spark benefits from its programming model that has RDDs, which overall shortens the required number of lines and number of classes. In contrast to Spark, MapReduce requires a higher number of classes because of the required two sets of jobs. The two sets of jobs consisted of two sets of Mapper and Reducer classes, which increase the overall number of lines in the implementation. The metrics in Table 1 prove that MASS is a better programming tool in terms of the required steps needed to
set up the environment, and Spark is a better programming tool in terms of the fewer number of classes and lines of code required in the implementation.

2.2.2 Execution Performance

The following screenshots present the execution performance results produced by MapReduce, Spark, and MASS implementations of the Range Search. The programs are executed using four worker nodes and 10000 input points.

**MapReduce.**

```bash
Elapsed time: 4 seconds
[sparon@chezme05 hadoop-2.7.3]$ ./getOutput.sh
Found 2 items
-rw-r--r-- 3 sparon sparon 0 2020-05-12 15:51 /user/sparon/output/_SUCCESS
-rw-r--r-- 3 sparon sparon 20173 2020-05-12 15:51 /user/sparon/output/part-00000
master 994,2643
master 948,2958
master 908,2853
master 910,2863
master 930,2967
master 857,3298
master 852,2889
master 856,2807
master 894,2856
master 991,1562
```

*Figure 1. Screenshot of MapReduce implementation of Range Search with four worker nodes.*

**Spark.**

```bash
20/05/25 14:28:27 INFO DAGScheduler: ResultStage 0 (collect at RangeSearch
Spark.java:184) finished in 5.372 s
20/05/25 14:28:27 INFO DAGScheduler: Job 0 finished: collect at RangeSearc
hSpark.java:184, took 5.422476 s
Elapsed time: 5 seconds

======== TOTAL INPUT NUMBER OF POINTS: 10000
======== Points found in requested range: 370
======== Found points in range:
313,385
258,305
255,307
255,377
259,355
265,371
270,358
265,368
268,378
309,334
303,309
318,318
318,331
347,331
```

*Figure 2. Screenshot of Spark implementation of Range Search with four worker nodes.*
MASS.

Figure 3. Screenshot of MASS implementation of Range Search with four worker nodes.

**Performance Comparison.** The Range Search implementation with the MASS library benefits from the fact that MASS allows maintaining the original dataset structure for the duration of computation. The data is not copied continuously or moved as in MapReduce and Spark. The same graph containing points on its vertices is used throughout the entire computation. Thus, MASS implementation produces better execution performance results in comparison with MapReduce and Spark applications. According to the performance results gathered during multiple tests (see Figure 4), the performance of MASS implementation increases by increasing the number of cluster nodes. Further, MapReduce produced better performance results using two and three worker nodes in comparison with Spark implementation. Yet, in contrast to Spark implementation, the execution performance of MapReduce implementation does not improve when the number of worker nodes is greater than three (see Figure 4).

Figure 4. Spark, MapReduce, and MASS implementations of Range Search execution performance with four worker nodes with 10000 input points.
3 Future Improvements

MapReduce and Spark
Both implementations require the input points to be sorted, duplicate points removed, and points partitioned into slices before passing the partitions for further computation. These initial steps slow down the performance of MapReduce and Spark implementations. The improvement would be to preprocess the input points and write an input file with pre-determined point partitions before the Range Search implementations start their execution. The preprocessing will decrease the number of MapReduce jobs and the number of Spark RDDs that should increase the overall execution performance of both applications.

MASS
The current GraphPlaces implementation in the MASS library made it easier to create a KD tree in comparison if only Places class would have been used. Yet, to migrate an Agent to a specific graph’s vertex, it is required to retrieve two types of vertex information, such as vertex’s unique identifier (ID) and GlobalKeyForKey. The vertex ID is an argument passed by a user when a vertex is added to the graph. The GlobalKeyForKey variable value is assigned when a vertex is created on a low level of GraphPlaces implementation by the MASS library. The following example demonstrates how the user should get the vertex place where an agent to migrate, MAASSBase.getGlobalIndexForKey (unique ID). This operation is not apparent to a user that just starts using MASS. It would improve the usability of MASS library’s GraphPlaces if a dedicated public function would return an index of the vertex that could be used as an argument to migrate() function. Another improvement would be the ability to spawn an Agent on a specific vertex without a tie to the parent’s vertex. The ability to swan a child agent on a different from parent’s vertex would eliminate the need to keep track of child or parent agent operations in the RangeSearchAgent class.

4 References


